CMPE 202 Individual Project
Navami Murthy Sanjaynagar
SJSU ID: 015278172

CREDIT CARD PROBLEM

1 - Describe what is the primary problem you try to solve.

The primary problem which I am trying to solve is to check whether a card is valid. If the card is valid, we should determine the card issuer using behavior pattern. And if the card doesn't correspond to any category, the specific credit card record should be considered as invalid.

Solved the problem using behavioral design patterns: Strategy Design Pattern and Chain of responsibility Design Pattern.

2 - Describe what are the secondary problems you try to solve (if there are any).

The secondary problem is how to deal with object creation mechanisms, trying to create objects suitable to validate the credit card type. In this problem statement, I am trying to extract the credit card data from a file. Three files need to be parsed– json, xml, csv and the output has to be a file of the same format as the input file, with each line showing the card number and the type of card (If the card is valid) or an error (If the card is invalid). After mapping the credit card to the respective sub class of the Credit card (can be Master Card, American Express, Visa, Discover), the following task is to create an instance of the credit class object as specified.

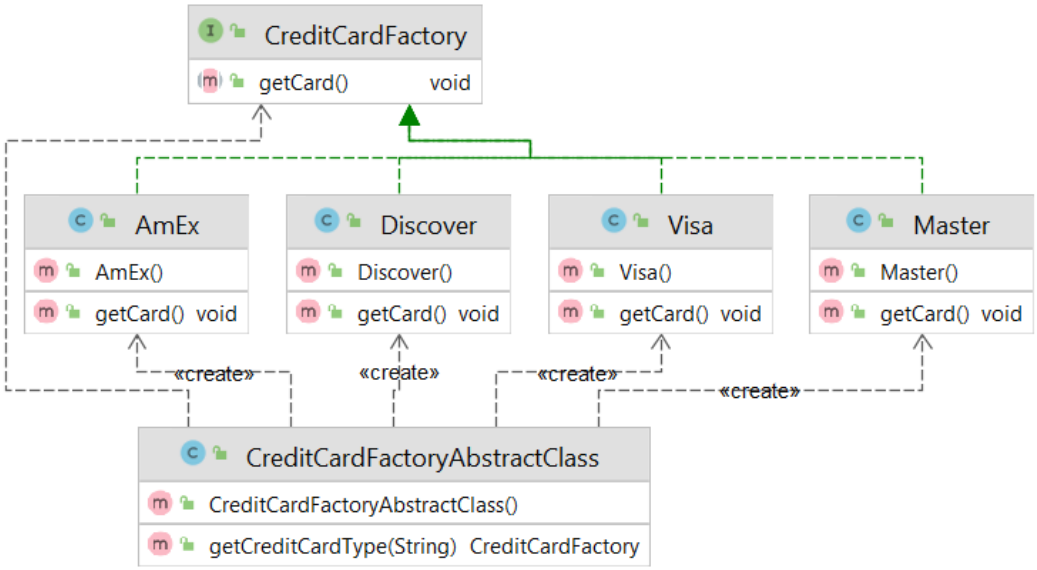
Solved this problem using creating design pattern: Factory Design Pattern

3 - Describe what design pattern(s) you use how (use plain text and diagrams).

Design patterns: Factory design pattern and Chain of responsibility design pattern.

a) Factory design pattern:

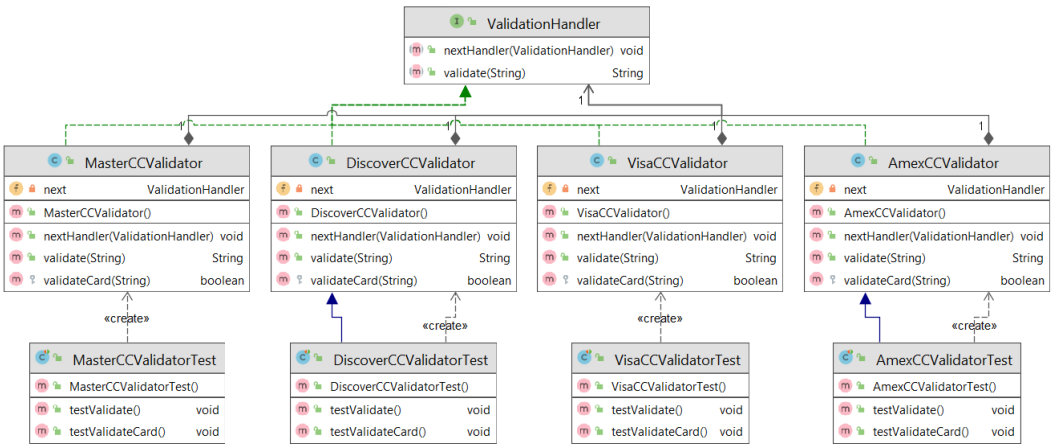
- ➢ Following the parsing of the csv file, we obtain the credit card number in each entry, which we must validate for each of the four credit card categories, and if it matches any of the specified regular expression patterns for each of the four categories, we must inform the parser that this credit card belongs to one of the four categories.
- ➢ The factory pattern can be used for this; it abstracts the credit card classes, but we construct an object of these credit cards using a factory using an abstract class (CreditCardFactoryAbstractClass) (CreditCardFactory).
- ➢ It generates a suitable object of corresponding credit card based on the string returned by the parser.



b) Chain of responsibility design pattern:

➢ After parsing the CSV file and obtaining credit card information, there should be a mechanism in place to verify the credit card number in order to determine which of the four categories it belongs to. ➢ The chain of responsibility pattern can be used to verify if a specific card number belongs to the current category, and if it doesn't, it can be moved on to the next category. The ValidationHandler interface controls the entire process by passing the credit card number to the four validators: AmExCCValidator, MasterCCValidator, VisaCCValidator, and DiscoverCCValidator.

➢ All the four validators implement the interface and pass on the card number to the next when it does not match with itself.



4- Describe the consequences of using this/these pattern(s).

Consequences of Factory Design Pattern:

Pros:

1) It decoupled the business logic of creation of a card creation classes from the actual logic of finding the credit card type in the parsers
2) It gives scope to add new credit card types in future
○ The creator is not tightly coupled to any ConcreteProduct.
3) Allows you to change the design of your application more readily.
○ Makes our code more robust, less coupled and easy to extend.
4) Provides abstraction between implementation and client classes through inheritance.

Cons:

1) Parsers might have to subclass the CreditCardFactory interface just to create a particular credit card Object.
○ The client now must deal with another point of evolution.
2) Makes code more difficult to read as all of your code is behind an abstraction that may in turn hide abstractions.
3) Sometimes making too many objects often can decrease performance.

Consequences of Chain of responsibility Design Pattern:

Pros:

1) It decouples the file parsers and its corresponding credit card validators.
○ Frees an object from knowing which another object handles a request
○ Both the receiver (file parsers) and the sender(validators) have no explicit knowledge of each other
2) Simplifies your object
○ It does not have to know the chain's structure to keep direct references to its members
○ Keeps a single reference to their successor
3) Gives you added flexibility in distributing responsibilities among objects. It allows you to add or remove responsibilities dynamically by changing the members or order of the chain.

Cons:

1) One drawback of this pattern is that the execution of the request is not guaranteed. It may fall off the end of the chain if no object handles it.
2) Also, that it is tedious to observe and debug at runtime.

Consequences of Strategy Design Pattern:

Pros:

- The Strategy pattern lets you indirectly alter the object's behavior at runtime by associating it with different sub-objects which can perform specific sub-tasks in different ways.
- The Strategy pattern lets you extract the varying behavior into a separate class hierarchy and combine the original classes into one, thereby reducing duplicate code.
- The Strategy pattern lets you isolate the code, internal data, and dependencies of various algorithms from the rest of the code. Various clients get a simple interface to execute the algorithms and switch them at runtime.
- You can swap algorithms used inside an object at runtime.
- You can introduce new strategies without having to change the context.

CMPE 202 Individual Project
Navami Murthy Sanjaynagar
SJSU ID: 015278172

CREDIT CARD PROBLEM PART 2

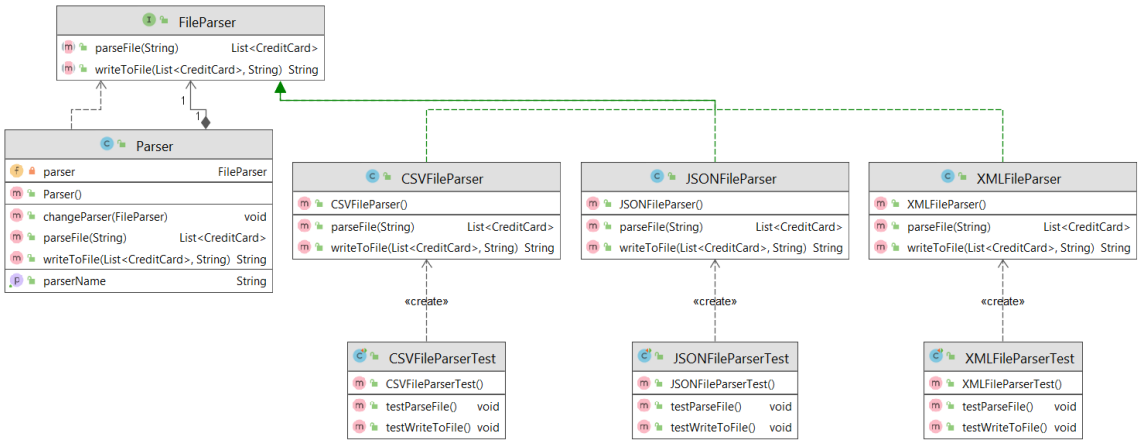
Extending the design from Part 1:

Strategy:

It is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Extending the above design to xml and json formats:

- ➢ To convert this file formatting to xml and json, I used the Strategy design pattern. It gives us the flexibility to add this feature to new file formats in the future.
- ➢ We can encapsulate interface (FileParser) information in a base class (CSVFileParser, JSONFileParser, XMLFileParser) and bury implementation details in derived classes (CSVFileParser, JSONFileParser, XMLFileParser) using this pattern.
- ➢ This entire design will allow to add a new credit card type/ new file format with ease without changing much of the code, just by creating classes and their corresponding validators.
- ➢ This design makes an abstraction between these modules and gives flexibility to the creators.



Thus, to solve the complete Credit Card problem, I used three design patterns –

1. Factory method
2. Strategy
3. Chain of responsibility

Complete UML diagram: